Reducing V&V cost of flight critical systems: myth or reality?

Guillaume P. Brat*

This paper presents an overview of NASA research program on the V&V of flight critical systems. Five years ago, NASA started an effort to reduce the cost and possibly increase the effectiveness of V&V for flight critical systems. It is the right time to take a look back and realize what progress has been made. This paper describes our overall approach and the tools introduced to address different phases of the software lifecycle. For example, we have improved testing by developing a statistical learning approach tor defining test cases. The tool automatically identifies possible unsafe conditions by analyzing outliers in output data; using an iterative learning process, it can then generate more test cases that represent potentially unsafe regions of operation. At the code level, we have developed and made available as open source a static analyzer for C and C++ programs called IKOS. We have shown that IKOS is very precise in the analysis of embedded C programs (very few false positives) and a bit less for regular C and C++ code. At the design level, in collaboration with our NRA partners, we have developed a suite of analysis tools for Simulink models. The analysis is done in a compositional framework for scalability.

Nomenclature

$V\&V$	Validation and Verification
C	The C programming language
$C++$	The C++ programming language
$NASA$	The National Aeronautics and Space Agency
$ARMD$	The NASA Aeronautic Research Mission Directorate
$NTSB$	The National Transportation Safety Board
FAA	The Federal Aviation Administration

I. Introduction

Nowadays software and systems represent more than half the cost of developing a new commercial transport aircraft. Moreover, in this cost category, verification and validation (V&V) drives 75% of the cost of developing software. This is mostly due to having to follow the DO-178B standard, which relies heavily on testing and imposes to achieve high MC/DC coverage. Leaving the V&V activities to this late in the process implies a high percentage of re-design (from re-coding to re-designing, and, in some cases, changing requirements). As shown in figure 1, 80-90% of faults are introduced in phases from requirements to code, and yet, the current process catches 96% of the faults in testing. It would be more efficient to catch errors when they are introduced (i.e., on the left side of the V). Recently with the new DO-178C standard, the FAA has opened the door to new V&V methods, which can be applied earlier in the process and help reduce the latency of finding errors thus avoiding (re-)development costs.

The NASA Aeronautic Research Mission Directorate (ARMD) has established a research program seeking to develop and inject advanced V&V techniques (such as formal methods) throughout the development process, from requirements through design and coding all the way to testing. This paper reports on our

*Robust Software Engineering Lead, Intelligent Systems Division, NASA Ames Research Center, Moffett Field, CA 94035, AIAA Member.

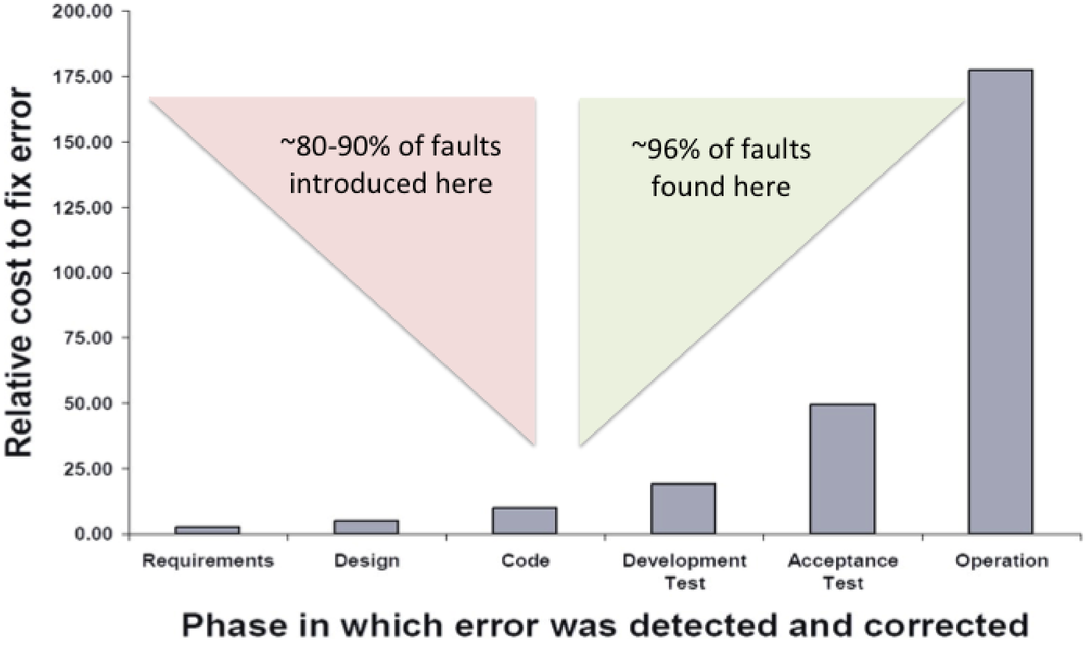


Figure 1. Cost of catching errors by phases of the lifecycle process.

effort to develop tools for the early stages of the development process (the so-called right side of the V) and their application on relevant case studies at NASA and in industry.

This paper is organized as follows. First, we described the V&V techniques we developed for design models, then the ones for code analysis, and then the ones for optimizing testing. We then described how the results of these tools can be incorporated into a safety case using our in-house tool. In our conclusions we also discuss future work.

II. Design Analysis

NASA is developing capabilities to automatically verify requirements on design models and to catch design errors as soon as they are introduced. Requirements are disambiguated and turned into formal requirements, which are then checked against design models to verify that the design meets the requirements of the system. NASA has implemented this capability in a tool called CoCoSim and is relying on compositional verification for managing scalability.

II.A. Requirement Verification

CoCoSim is a verification tool targeting design models. The overall goal of CoCoSim is to verify that system requirements (especially safety ones) hold for the design models of the system. The implementation of the tool relies on a more general theory about using Horn clauses for verifcation purposes.[6] CoCoSim can handle:

- LUSTRE models,
- Simulink models, except for embedded MatLab code (which is in the process of being addressed but is not yet available)
- SCADE models, and,

- Stateflow models.

Behind the scene, CoCoSim transforms models and requirements into LUSTRE programs, which are analyzed using the Theory of Horn clauses. Therefore, LUSTRE and SCADE models are straightforward to handle. For Simulink, the analysis process requires a user to represent their requirements as Simulink models; these models are then combined with the Simulink models representing the system and transform into a LUSTRE program suitable for analysis. CoCoSim is fully connected with the Simulink environment to facilitate the transformation of requirements into Simulink models. The error reporting is also integrated in the Simulink environment and allows the playback of error traces as Simulink simulations. In summary, CoCoSim brings model checking of safety requirements for industrial design models such as Simulink models (often used for designing flight controllers) and SCADE models (often used for designing engine controllers). CoCoSim also supports compositional verification for large models (see next section).

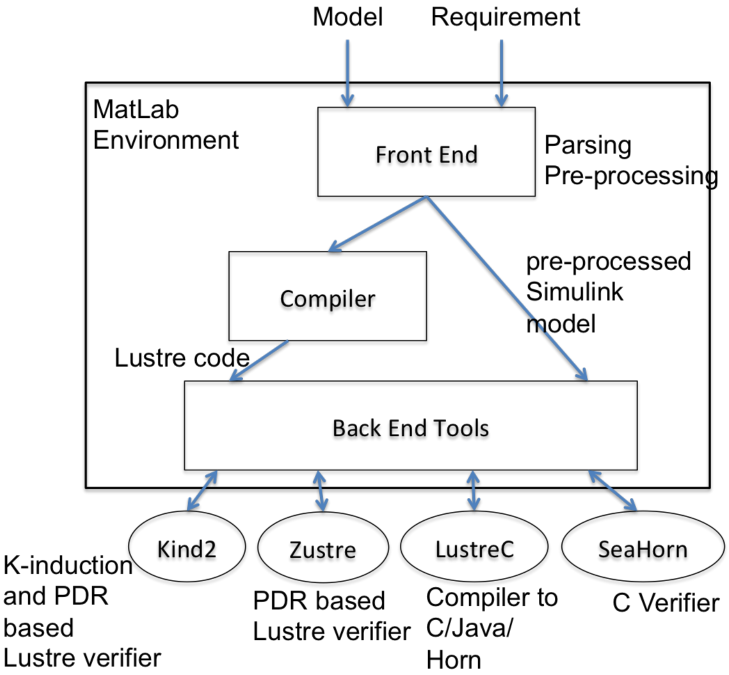


Figure 2. CoCoSim Architecture.

As shown in figure 2, CoCoSim can interface to several analysis tools, which can analyze Lustre code or C code. This allows CoCoSim to analyze Simulink models as well as the C code generated from these models. Since the requirements are represented as Simulink (observer) models, they are automatically translated into Lustre or C and are thus analyzable in the context of the system design model.

II.B. Compositional Verification

It is well known that formal methods can experience scalability problems even on design models. To address this problem, our program has been relying on compositional verification. We are using an assume-guarantee style of compositional verification and we have illustrated its use on a case study describing a scaled-down version of a generic transport aircraft, the TCM or Transport Class Model.[1] We started with existing Simulink models for the TCM and derived requirements for the TCM from requirements for the Boeing 737. Here is an example of one of the 20 requirements we developed for the TCM:

GUIDE-210	*If the FPA control and the altitude control are both selected, the FPA control will disengage and the altitude control will engage once the plane is within 200 ft of the commanded altitude.*

As shown in table 1, we formally verified 17 of the 20 system-level guidance requirements. Ten of the requirements could be formally verified directly on the model, with no decomposition. One property was

Table 1. Verified requirements for the TCM example.

Verification	Number
Direct (K-induction Model Checking)	10
Direct (Bounded Model Checking + testing)	1
Compositionally (K-induction Model Checking)	6
DNF (no relation to model)	3

verified directly using bounded model checking and testing. An additional three properties were unverifiable because the TCM model did not include the specified behaviors. A total of six requirements required compositional reasoning.

As an example, the property

GUIDE-120	*The Guidance system shall be capable of climbing at a defined rate, to be limited by min. and max. engine performance and airspeeds*

was broken down into four sub-requirements:

ALT-1	*If not in Altitude Control Mode, the Altitude control is not engaged.*
GUIDE-180	*The FPA control shall engage when the FPA control mode is selected and when there is no manual pitch or manual roll command from the stick.*
ALT-2	*If not engaged, the Altitude control will not issue a command for the flight path angle.*
FPA-1	*If engaged, the FPA control will issue pitch commands to maintain a FPA.*

and required the following two assumptions (generated manually) for complete proof:

The sensor signals given to the guidance are correct.
Output commands from the GNC system are implemented correctly by the working effectors.

III. Code Analysis

For code analysis, NASA has developed a static analysis tool called IKOS, which targets buffer overruns, integer underflow/overflow, and uninitialized variables and pointers, and a method to analyze floating-point operations, which will be implemented in the FramaC analysis framework.

Static code analysis is a verification technique that identifies errors that can happen at runtime (called runtime errors) in software without having to execute the program; some analyzers, like IKOS, can also certify the absence of run-time errors. Using the formal semantic of a programming language (C/C++ in the case of IKOS), this technique analyzes the source code of a program looking for errors of a certain type (see below). Static code analysis has been around for many years but commercial analyzers have failed to deliver precision (very little false positives) and scalability for C and C++.

NASA has developed the IKOS[2] framework with the underlying philosophy that precision and scalability can be achieved by customizing analyzers to families of software (e.g., embedded code). IKOS is thus designed to facilitate the quick development of analyzers as long as they are based on the Theory of Abstract Interpretation.[3] To achieve this, IKOS offers a library of Abstract Interpretation components, which can be easily assembled and targeted at a given language (e.g., C or C++). Using these components produces sound analyzers, i.e., analyzers that guarantee to not report an error when there is none.

In order to illustrate the capabilities of the IKOS library, NASA has developed an IKOS analyzer that aims at proving the absence of the following runtime errors in C and C++ programs:

Buffer overrun	attempting to access an array outside its bounds (before or after the memory allocated for the array).
Integer division by zero	attempting to divide an integer by zero.,
Null pointer dereference	attempting to access the content referenced by a pointer which is however set to the null pointer (no content)
Read of uninitialized integer variables	attempting to read the contents of variables that haven't been set yet (erroneous value).

In addition, IKOS can also report any violation of user-defined properties in the form of user-added assert statements in code. This operating mode is not recommended as it requires manual instrumentation of the code, which can be error prone.

IKOS was developed for flight critical embedded systems in aviation. Originally, we mostly targeted avionic systems (flight control systems, flight management systems, engine control, and so on). These types of code are often developed from design models written in Simulink or SCADE. Sometimes the code is automatically generated from the models. In general, the code targets on-board computers and have the general characteristics of real-time, embedded systems. We do not analyze the code for real-time issues but the real-time aspect has an impact on the shape of the code. Thus memory tends to be allocated upfront in an initialization phase; data structures tend to be large but fairly shallow (not many levels of indirections). These characteristics are favorable for static analysis. However embedded code can also have many function pointers, which usually introduce imprecisions in static analysis. As expected we have been unable to try IKOS on commercial flight systems. Therefore, we focused on NASA-developed controllers and open source controllers mainly for UAVs for validating IKOS. Our list of avionic codes is as follows:

AeroQuad	open source software (and hardware) system, which can be used to build remote controlled quadcopters and other types of multicopter configurations. It consists of roughly 167 KLOC of C code (headers included) and it targets an Arduino micro-controller.
Cornell	open source version of an autonomous stability system for a co-axial helicopter developed at Cornell University as a class project. It is our smallest example of a flight control system on C
DIY AutoPilot	open source software dedicated to the control of drones. The systems using this autopilot include ArduPilot for example. It consists of 162 KLOC of C code (including headers).
Gen2 ARINC	NASA prototype for an adaptive controller for an F-15 aircraft. The neural net controller can learn new flight dynamics when an aircraft gets damaged during flight and take over from the traditional controller. The 13KLOC of C code are automatically generated from Simulink models and they target an ARINC processor.
FlightZ	flight simulation environment that includes the Gen2 controller.
MNAV-autopilot	open source software for controlling all kinds of robotics applications, but our analysis focuses on the version for drones. It consists of 159 KLOC of C code.
Paparazzi UAV	open-source drone hardware and software project encompassing autopilot systems and ground station software for multicopters/multirotors, fixed-wing, helicopters and hybrid aircraft. Our analysis focuses on a 22KLOC C flight controller for a microjet.

Table 2 shows that IKOS is doing really well on embedded code. IKOS does not produce any warnings on four of our case studies and achieves a general level of less than 7% of warnings for the other two software. Even though the case studies are small (they're core flight control software), some of them (e.g., Paparazzi) contain lots of function calls. Yet IKOS does well because it uses dynamic in-lining in the analysis. The real question is: why are we not achieving perfect precision on these two software?

Table 2. Buffer overruns results

name	LOC	BC func	BC instrs	num checks	ok	error	warnings	unreachable
aeroquad	1̃67K	159	4,634	200	200	0	0 - 0%	0
cornell	439	8	469	76	64	0	0 - 0%	12
diy-autopilot	1̃62K	237	5,861	122	122	0	0 - 0%	0
flightz	91K	197	14,501	4726	4103	12	529 -11 %	82
gen2-arinc	1̃3K	96	5,340	292	254	0	20 - 7%	18
mnav-autopilot	1̃59K	33	2,145	742	694	0	48 - 6%	0
paparazzi-microjet	2̃2K	99	4,436	732	700	0	0 - 0%	32

For Gen2, the reason is because it implements a neural net controller, which has a very different structure than embedded code; it basically is implemented as a matrix. This is typically a problematic data structure for static analyzers. To complicate matters further, Gen2, which is a research code rather than a commercial-strength software, stores the size of the matrix at the beginning of the memory allocated for the matrix; therefore, the static analyzer needs not only to keep track of the memory size and offset but also on some of the contents. In this case IKOS does pretty well since it is able to pick up the size and take it into account in the analysis. This is a double-edge sword since it might mean that occasionally IKOS might waste time analyzing content of arrays or matrices.

For the Paparazzi code, most warnings are due to imprecisions in handling variables in loops. The current version of IKOS uses mostly the interval domain, which causes IKOS to lose the connection between condition variables and index variables. We are planning to inject a "homeopathic" dose of relational domains in IKOS for addressing these types of code constructs.

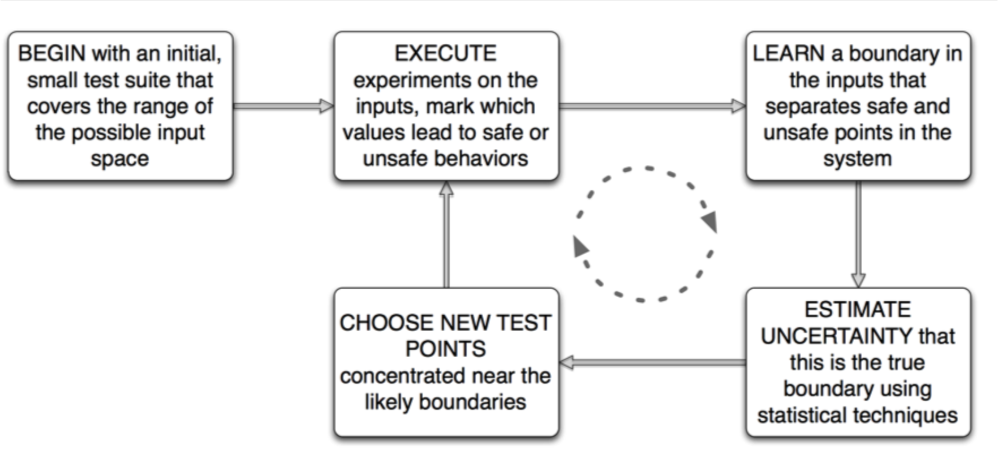
NASA has also developed static analysis capabilities for analyzing floating-point computations, especially to symbolically compute error bounds when using IEEE floating-point arithmetic. This work has been applied to the Compact Positioning Reporting (CPR) algorithm used in Automatic Dependent Surveillance-Broadcast (ADS-B) and it actually found problems in CPR. All the finding have been delivered to the RTCA-SC-186/EUROCAE Working Group 51 in charge of standards for ADS-B. The work has resulted in a formally verified C implementation of the CPR algorithm. This work builds on previous publications on analyzing real numbers.[9]

Requirement verification using static analysis is possible using the SeaHorn[7] tool mentioned in the section on Design Analysis using the CoCoSim tool as described in figure 2. Requirements are expressed as Simulink (observer) models and combined with the system models. The combined models are then translated into C and verified using SeaHorn. This enables requirement verification from design to code.

IV. Advanced Testing

Our main objective is to enable early verification and validation to avoid the cost associated with finding errors late in the process, i.e., in testing. However, our goal is not to eliminate testing, but rather enable developers to "fly through testing". We still think that testing has an important role to play in the development process. For one thing, it brings a feeling of comfort. We often hear about "flying what we test". This saying shows that operators (pilots or controllers) are more comfortable operating a system that has been through a series of test. At least for these reasons testing should not be eliminated. However, it should and can be improved so that it helps identify unsafe regions of operations. This is precisely why we designed the Margins tool. Margins aims at identifying, through statistical learning, safety boundaries for non-linear aeronautical systems in which inputs and outputs can be time series (e.g., trajectories).

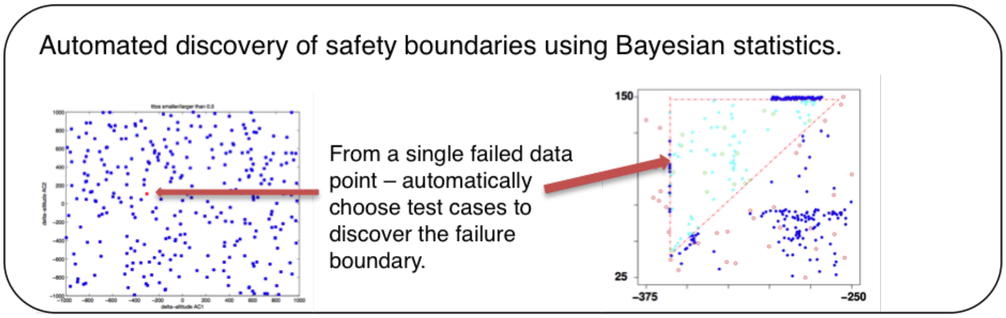
In figure 3 we describe the typical use of the Margins tool.[8] The first step is to create an initial test suite that covers well the input space of a system/software. This can be done using any traditional technique such as N-combinatorial coverage of the input variables. Then comes the iteration phase, which starts with executing the experiments and classifying inputs that lead to safe or unsafe behaviors. The next step is to learn a boundary in the input space that separates the safe and unsafe outputs; this is done using Bayesian statistics. Then, Margins calculates the uncertainty that the identified region is truly the unsafe region. If we are satisfy that the uncertainty is small enough, we can stop at this point. If not, then Margins uses the information it learned to identify new inputs (around the already identified dangerous inputs) and refine the safe-unsafe classification.


BEGIN with an initial, small test suite that covers the range of the possible input space
EXECUTE experiments on the inputs, mark which values lead to safe or unsafe behaviors
LEARN a boundary in the inputs that separates safe and unsafe points in the system
ESTIMATE UNCERTAINTY that this is the true boundary using statistical techniques
CHOOSE NEW TEST POINTS concentrated near the likely boundaries

Stop when the uncertainty is small enough or at maximum time (with the smallest statistical uncertainty possible in that time)

Figure 3. Process used by the Margins tool.

In figure 4 we illustrate how the Margins tool can identify safe boundaries of operations for a given software system. The principle is simple. First, Margins is used to identify outlier (system) outputs and their corresponding (system) inputs. After manual inspection, the outlier is classified as safe or unsafe. If it is unsafe, Margins is instructed to learn (through the use of Bayesian statistics) the whole unsafe region around the outlier point and the corresponding input set. As a result of the analysis, Margins has learned potential dangerous inputs that need to be avoided during operations.


Automated discovery of safety boundaries using Bayesian statistics.

From a single failed data point – automatically choose test cases to discover the failure boundary.

Figure 4. Identifying safe boundaries using Margins.

V. Evidence-based Certification

As described in this paper, NASA has developed many tools that provide evidences that risks have been retired at different stages of the lifecycle for a given system. If DO-178C, and its DO-333 supplement, now provides a way of getting certification credits for the use of formal methods, it still doesn't really connect the results of formal method tools to specific risks in the system. This creates the potential for misleading safety claims. Therefore, NASA has been working on tools that can create explicit safety claims in the form of assurance cases. We use the term assurance cases rather than safety (or dependability) cases because it is broader and can also speak about the performance of the system.

For us, an assurance case is a set of assurance claims connected to a body of evidences through a structured argument, to provide a comprehensive, defensible and valid justification that a system meets its assurance requirements for a given application in a defined operating environment. In practice, it stores the argument as a structured database of assurance assets with tracing relations and thus provides a means for integrating safety and mission assurance (S&MA) information. An assurance case integrates a variety of assurance assets, tool capabilities, and information from standards. As shown in figure 5 an assurance case provides a least the following services: assurances that system attributes meet requirements, a dashboard for decision making, support for stakeholder viewpoints and report generation, justifications of compliance with standards, procedural requirements, and advices, i.e., assurance options (e.g., tool selection / preconditions) to meet assurance goals based on compliance needs.

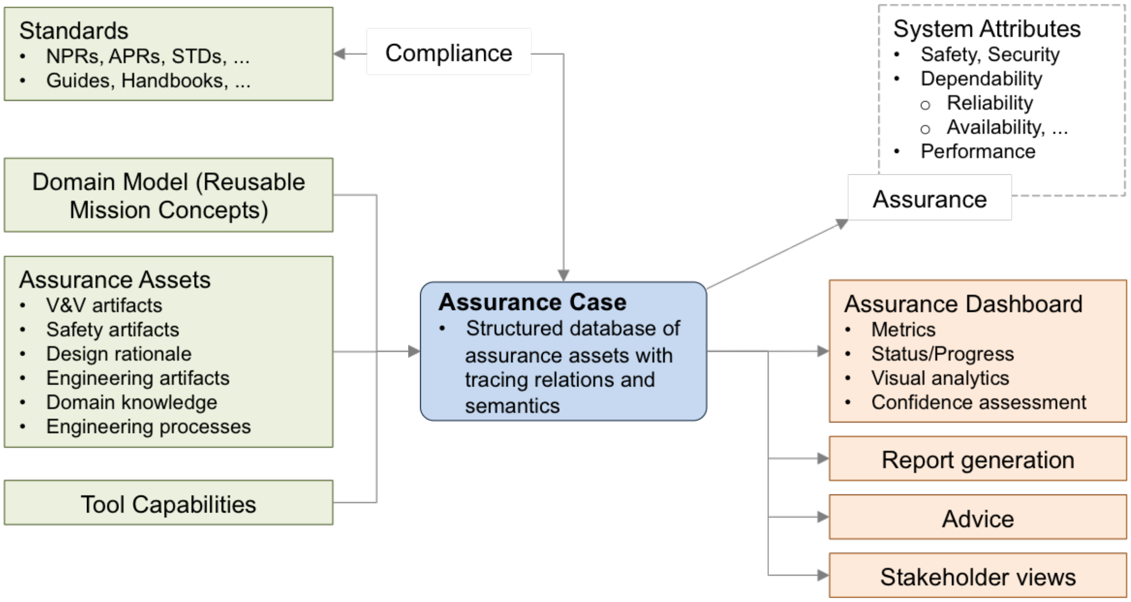


Figure 5. Structure and contents of an assurance case.

We are implementing these services in a tool called AdvoCATE,[4,5] which is built in the Eclipse environment and offers services to create, visualize and inspect assurance cases. Figure 6 shows how results of formal methods are integrated into AdvoCATE. Note that AdvoCATE captures not only the results of applying a formal tool but also the formal inputs used by the tool (e.g., formal requirements or assumptions) and the method used to derive the results (e.g., compositional reasoning). This enables comprehensive reviews of assurance claims, including disclosures of the methods used during the analysis.

VI. Conclusion

In this paper, we have described parts of the work done by NASA to address the high cost associated with current V&V processes in civil aviation. We have described many tools that can be applied at early phases of the lifecycle, thus enabling to catch errors closer to where they have been introduced. We believe that a systematic application of our tools will enable industry to reduce their cost by avoiding catching errors late in the process (at testing or even acceptance testing), which yields additional re-design or re-coding costs. We presented techniques that can be applied at design time, coding time, or even can help make testing more thorough. We also believe that the risks retired by these techniques are better represented in assurance cases rather than the usual DO-178C standard process. Therefore, we created a tool that not only helps in creating, visualizing, and inspecting assurance cases, but also can be used as an index to search through a risk-informed database and reason about safety.

Our tool suite is by no means complete. It is at least missing a critical component to manage requirements. To take full advantage of the tools presented here, one needs to have formal requirements. It would also enable early analysis of requirements, which NTSB has been advocating for many years; many accidents can be linked to problems in requirements. The notion of formal requirements is not new. There exist many languages (too many to cite them) for creating formal requirements. However, none of them are used in practice in industry because their notations are too scary (e.g., too mathematical) or not expressive enough (e.g., too graphical). Our solution aims at creating tools that can take textual requirements and turn them into formal ones. Recent advances in natural language recognition have given us hope that it is possible. Yet,

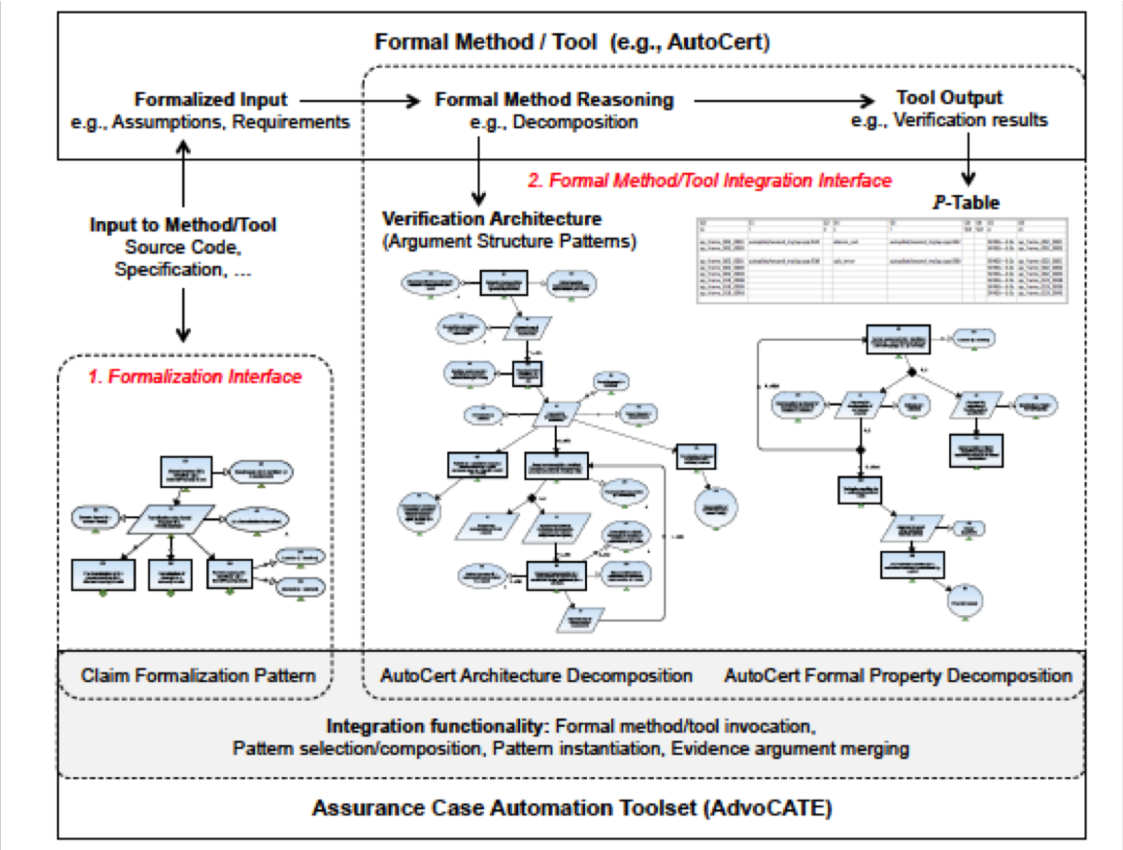

Formal Method / Tool (e.g., AutoCert)
Formalized Input
e.g., Assumptions, Requirements
Formal Method Reasoning
e.g., Decomposition
Tool Output
e.g., Verification results
2. Formal Method/Tool Integration Interface
P-Table
Input to Method/Tool
Source Code,
Specification, ...
Verification Architecture
(Argument Structure Patterns)
1. Formalization Interface
Claim Formalization Pattern
AutoCert Architecture Decomposition
AutoCert Formal Property Decomposition
Integration functionality: Formal method/tool invocation,
Pattern selection/composition, Pattern instantiation, Evidence argument merging
Assurance Case Automation Toolset (AdvoCATE)

Figure 6. Incorporating formal methods in AdvoCATE.

we do not believe that it is the full answer. By nature requirements are ambiguous; most of us are better at expressing what we do not want rather than what we want. As humans we love living with ambiguities; it makes our life simpler. However, when it comes to verification ambiguity is a bad thing. Therefore, we are working on a tool that will establish a dialogue with the requirement developers and guide them in developing unambiguous formal requirements. This work is still one or two years away from being usable in practice.

The other big frontier in front of the V&V community is dealing with autonomous systems. Autonomy will bring new challenges like verifying and validating learning capabilities be it from planning & scheduling, or machine learning, or reinforcement learning. For these systems we should not commit the same mistake as with past and current systems and rely only on testing. We need to find methods that can be applied early during the development of those systems. It will require us to have a deep understanding of the mathematical underpinnings of these techniques and add capabilities to reason about uncertainties. The other challenging aspect of autonomy is in the use of many sensors, which results in systems with a very large input space. Even if we rely on testing mainly, this will present us with a huge challenge in terms of covering the input space.

Acknowledgments

Many researchers at NASA have contributed to the work described on this paper such as Cesar Munoz, Ewen Denney, Ganesh Pai, Ian Whiteside, Yuning He, Misty Davies, Temesghen Kahsai, Nija Shi, Maxime Arthaud, David Bushnell, and Dimitra Giannakopoulou as well non-NASA contributors like Pierre-Loic Garoche, Jorge Navas, Arie Gurfinkel, Joseph Pohl, Arnaud Venet, and Falk Howar.

References

[1] G. Brat, D. Bushnell, M. Davies, D. Giannakopoulou, F. Howar, and T. Kahsai. Verifying the safety of a flight-critical system. In *International Symposium on Formal Methods*, pages 308–324. Springer International Publishing, 2015.

[2] G. Brat, J. Navas, N. Shi, and A. Venet. IKOS: a Framework for Static Analysis based on Abstract Interpretation. In *SEFM*, 2014.

[3] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[4] E. Denney, G. Pai, and J. Pohl. Advocate: An assurance case automation toolset. In *Computer Safety, Reliability, and Security - SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings*, pages 8–21, 2012.

[5] E. Denney, G. J. Pai, and I. Whiteside. Formal foundations for hierarchical safety cases. In *16th IEEE International Symposium on High Assurance Systems Engineering, HASE 2015, Daytona Beach, FL, USA, January 8-10, 2015*, pages 52–59, 2015.

[6] P.-L. Garoche, A. Gurfinkel, and T. Kahsai. Synthesizing modular invariants for synchronous code. *arXiv preprint arXiv:1412.1152*, 2014.

[7] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer International Publishing, 2015.

[8] Y. He and M. D. Davies. Bayesian statistics and uncertainty quantification for safety boundary analysis in complex systems. 2014.

[9] M. Moscato, C. Muñoz, and A. Smith. Affine arithmetic and applications to real-number proving. In C. Urban and X. Zhang, editors, *Proceedings of the 6th International Conference on Interactive Theorem Proving (ITP 2015)*, volume 9236 of *Lecture Notes in Computer Science*, pages 294–309, Nanjing, China, August 2015. Springer.